Articulation_points

```
vector<vector<int>> d, comp;
vector<int> tin, up;
vector<pair<int, bool>> bypass;
int time1;

void build_comp(int vertex) {
    comp.push_back({vertex});
    while (bypass.back().first != vertex) {
        if (bypass.back().second) {
            comp.back().push_back(bypass.back().first);
        }
        bypass.pop_back();
    }
}

void DFS_articulation(int vertex, int parent) {
    tin[vertex] = up[vertex] = time1++;
    bypass.emplace_back(vertex, true);
    for (int q : d[vertex]) {
        if (tin[q] == -1) {
            bypass.emplace_back(vertex, false);
            DFS_articulation(q, vertex);
            if (up[q] >= tin[vertex]) {
                build_comp(vertex);
            }
            up[vertex] = min(up[vertex], up[q]);
        } else if (q != parent) {
            up[vertex] = min(up[vertex], tin[q]);
        }
    }
}

vector<vector<int>> g;

int make_articulation(int n) {
    tin.assign(n, -1), up.assign(n, -1);
    comp = {}, bypass = {}, time1 = 0;
    DFS_articulation(0, -1);
    int k = (int)comp.size();
    g.assign(n+k, {});
    for (int q = 0; q < k; q++) {
        for (int q1 : comp[q]) {
            g[q+n].push_back(q1);
            g[q1].push_back(q+n);
        }
    }
    return k;
}
```

Dinitz

```
struct Edge {
    int x, y, c, f;
};

struct Flow {
    vector<vector<int>> gf;
    vector<Edge> edges;

    explicit Flow(int n) {
        gf.assign(n, {});
    }
```

```
        void add_edge(int x, int y, int c, bool directed) {
            gf[x].push_back((int)edges.size());
            edges.emplace_back(x, y, c, 0);
            gf[y].push_back((int)edges.size());
            edges.emplace_back(y, x, (1-directed)*c, 0);
        }

        vector<int> layer, ind;

        bool build_layers(int x, int y) {
            layer.assign(gf.size(), -1);
            ind.assign(gf.size(), 0);
            queue<int> a;
            a.push(x);
            layer[x] = 0;
            while (!a.empty()) {
                int q = a.front();
                if (q == y) {
                    return true;
                }
                a.pop();
                for (int q1_ : gf[q]) {
                    Edge& q1 = edges[q1_];
                    if (q1.c == q1.f) {
                        continue;
                    }
                    if (layer[q1.y] == -1) {
                        a.push(q1.y);
                        layer[q1.y] = layer[q1.x]+1;
                    }
                }
            }
            return layer[y] != -1;
        }

        int push(int x, int y, int min1) {
            if (x == y) {
                return min1;
            }
            int ans = 0;
            for (; ind[x] < gf[x].size(); ind[x]++) {
                int num = gf[x][ind[x]];
                Edge& q = edges[num];
                if (layer[q.y] != layer[q.x]+1 || q.f == q.c) {
                    continue;
                }
                int pushed = push(q.y, y, min(min1, q.c-q.f));
                edges[num].f += pushed;
                edges[num ^ 1].f -= pushed;
                ans += pushed, min1 -= pushed;
                if (min1 == 0) {
                    return ans;
                }
            }
            return ans;
        }

        void build_flow(int x, int y) {
            while (build_layers(x, y)) {
                push(x, y, INF);
            }
        }
        int max_flow(int x, int y) {
            build_flow(x, y);
            int ans = 0;
            for (int q : gf[x]) {
                ans += edges[q].f;
            }
            return ans;
        }
    };
```

Ford_Fulkerson

```
    struct Edge {
        int x, y, c, f = 0;
    };

    vector<vector<Edge>> d;
    vector<vector<int>> gf;
    vector<Edge> e;
    vector<bool> was;

    int push(int vertex, int x, int y) {
        if (vertex == y) {
            return x;
        }
        was[vertex] = true;
        for (int q : gf[vertex]) {
            Edge &q1 = e[q];
            if (!was[q1.y] && q1.c-q1.f > 0) {
                int x1 = push(q1.y, min(x, q1.c-q1.f), y);
                if (x1 != -1) {
                    q1.f += x1, e[q ^ 1].f -= x1;
                    return x1;
                }
            }
        }
        return -1;
    }

    bool cancel_way(int vertex, int y) {
        if (vertex == y) {
            return true;
        }
        was[vertex] = true;
        for (int q : gf[vertex]) {
            Edge &q1 = e[q];
            if (!was[q1.y] && q1.f > 0 && cancel_way(q1.y, y)) {
                q1.f--, e[q ^ 1].f++;
                return true;
            }
        }
        return false;
    }

    void cancel(int ind, int x, int y) {
        if (e[ind].f == 0) {
            return;
        }
        e[ind].c--, e[ind].f--;
        was.assign(d.size(), false);
        assert(cancel_way(x, e[ind].x));
        was.assign(d.size(), false);
        assert(cancel_way(e[ind].y, y));
    }

    void build_flow() {
        int n = (int)d.size();
        gf.assign(n, {}), e = {};
        for (int q = 0; q < n; q++) {
            for (Edge q1 : d[q]) {
                gf[q1.x].push_back((int)e.size());
                e.emplace_back(q1.x, q1.y, q1.c);
                gf[q1.y].push_back((int)e.size());
                e.emplace_back(q1.y, q1.x, 0);
            }
        }
    }

    int max_flow(int x, int y) {
        int n = (int)d.size();
        int size = 0;
        while (size != -1) {
            was.assign(n, false);
            size = push(x, INF, y);
        }
        int ans = 0;
        for (int q : gf[x]) {
            ans += e[q].f;
        }
        return ans;
    }
```

Min_cost

```
    struct Edge {
        int x, y, c, f, v;
    };

    struct Flow {
        vector<vector<int>> gf;
        vector<Edge> edges;

        explicit Flow(int n) {
            gf.assign(n, {});
        }

        void add_edge(int x, int y, int c, int v) {
            gf[x].push_back((int)edges.size());
            edges.emplace_back(x, y, c, 0, v);
            gf[y].push_back((int)edges.size());
            edges.emplace_back(y, x, 0, 0, -v);
        }

        vector<int> dists;

        void build_dists(int x) {
            int n = (int)gf.size();
            dists.assign(n, INF);
            vector<bool> taken(n, false);
            queue<int> a;
            dists[x] = 0, taken[x] = true;
            a.push(x);
            while (!a.empty()) {
                int q = a.front();
                a.pop();
                taken[q] = false;
                for (int q1 : gf[q]) {
                    Edge& e = edges[q1];
                    if (e.f != e.c && dists[e.y] > dists[q]+e.v) {
                        dists[e.y] = dists[q]+e.v;
                        if (!taken[e.y]) {
                            taken[e.y] = true;
                            a.push(e.y);
                        }
                    }
                }
            }
        }

        bool push(int x, int y) {
            int n = (int)gf.size();
            vector<int> will(n, INF), parents(n, -1);
            priority_queue<p> a;
            will[x] = 0;
            a.emplace(0, x);
            while (!a.empty()) {
                int len = -a.top().first, q = a.top().second;
                a.pop();
                if (len != will[q]) {
                    continue;
                }
                for (int q1 : gf[q]) {
                    Edge& e = edges[q1];
                    int will_dist = len+e.v+dists[e.x]-dists[e.y];
                    if (e.f != e.c && will[e.y] > will_dist) {
                        will[e.y] = will_dist, parents[e.y] = q1;
                        a.emplace(-will_dist, e.y);
                    }
                }
            }
            if (will[y] == INF) {
                return false;
            }
            while (x != y) {
                edges[parents[y]].f++;
                edges[parents[y] ^ 1].f--;
                y = edges[parents[y]].x;
            }
            for (int q = 0; q < n; q++) {
                will[q] -= dists[x]-dists[q];
            }
            dists = will;
            return true;
        }

        void build_flow(int x, int y, int k) {
            build_dists(x);
            for (int q = 0; q < k && push(x, y); q++);
        }

        int min_cost(int x, int y, int k = INF) {
            build_flow(x, y, k);
            int ans = 0;
            for (int q = 0; q < edges.size(); q += 2) {
                ans += edges[q].f*edges[q].v;
            }
            return ans;
        }

        vector<vector<int>> ways;

        void find_way(int x, int y) {
```

```
        if (x == y) {
            return;
        }
        for (int q1 : gf[x]) {
            Edge& q = edges[q1];
            if (q.f > 0) {
                find_way(q.y, y);
                edges[q1].f--, edges[q1 ^ 1].f++;
                ways.back().push_back(q1);
                return;
            }
        }
    }

    void decompose(int x, int y) {
        ways = {};
        vector<Edge> was_edges = edges;
        int k = 0;
        for (int q : gf[x]) {
            k += edges[q].f;
        }
        for (int q = 0; q < k; q++) {
            ways.emplace_back();
            find_way(x, y);
            reverse(ways.back().begin(), ways.back().end());
        }
        edges = was_edges;
    }
};
```

Kun

```
vector<vector<int>> d;
vector<int> pa, pb;
vector<bool> was_a, was_b;

bool find_chain(int vertex) {
    was_a[vertex] = true;
    for (int q : d[vertex]) {
        if (pb[q] == -1 || !was_a[pb[q]] && find_chain(pb[q])) {
            pa[vertex] = q, pb[q] = vertex;
            return true;
        }
    }
    return false;
}

mt19937 randint(17957179);

void Kun(int n, int m) {
    pa.assign(n, -1), pb.assign(n+m, -1);
    vector<int> perm(n);
    iota(perm.begin(), perm.end(), 0);
    shuffle(perm.begin(), perm.end(), randint);
    bool flag = true;
    while (flag) {
        flag = false;
        was_a.assign(n, false);
        for (int q : perm) {
            if (pa[q] == -1 && find_chain(q)) {
                flag = true;
            }
        }
    }
    }

    int max_matching(int n, int m) {
        Kun(n, m);
        return n-count(pa.begin(), pa.end(), -1);
    }

    void DFS_L_minus(int vertex) {
        was_a[vertex] = true;
        for (int q : d[vertex]) {
            if (q == pa[vertex] || was_b[q]) {
                continue;
            }
            was_b[q] = true;
            if (pb[q] != -1 && !was_a[pb[q]]) {
                DFS_L_minus(pb[q]);
            }
        }
    }

    int independent_set(int n, int m) {
        Kun(n, m);
        was_a.assign(n, false), was_b.assign(m, false);
        for (int q = 0; q < n; q++) {
            if (pa[q] == -1) {
                DFS_L_minus(q);
            }
        }
        return count(was_a.begin(), was_a.end(), true)+count(was_b.begin(), was_b.end(), false);
    }

    int paths_splitting(int n) {
        return n-max_matching(n, n);
    }

    void DFS_reachable(int vertex) {
        was_a[vertex] = true;
        for (int q : d[vertex]) {
            if (!was_a[q]) {
                DFS_reachable(q);
            }
        }
    }

    void make_transitive_closure(int n) {
        for (int q = 0; q < n; q++) {
            was_a.assign(n, false);
            DFS_reachable(q);
            d[q] = {};
            for (int q1 = 0; q1 < n; q1++) {
                if (q != q1 && was_a[q1]) {
                    d[q].push_back(q1);
                }
            }
        }
    }

    int max_antichain(int n) {
        make_transitive_closure(n);
        return independent_set(n, n)-n; // was_a && !was_b
    }
```

HLD

```
vector<vector<int>> d;
vector<int> parents, sizes, height;

void DFS_sizes(int vertex, int parent, int h) {
    parents[vertex] = parent;
    sizes[vertex] = 1, height[vertex] = h;
    for (int q : d[vertex]) {
        if (q != parent) {
            DFS_sizes(q, vertex, h+1);
            sizes[vertex] += sizes[q];
        }
    }
    ranges::sort(d[vertex], {}, [](int x) {return -sizes[x];});
}

vector<int> up, tin, tout;
vector<int> order;

void DFS_create(int vertex, int parent, bool first) {
    tin[vertex] = (int)order.size();
    order.push_back(vertex);
    up[vertex] = (first ? up[parent] : vertex);
    first = true;
    for (int q : d[vertex]) {
        if (q != parent) {
            DFS_create(q, vertex, first);
            first = false;
        }
    }
    tout[vertex] = (int)order.size();
}

DO do_arr({});

void build_HLD(int vertex, vector<int>& a) {
    int n = (int)a.size();
    parents.assign(n, -1), sizes.assign(n, -1), height.assign(n, -1);
    DFS_sizes(vertex, -1, 0);
    up.assign(n, -1), tin.assign(n, -1), tout.assign(n, -1), order = {};
    DFS_create(vertex, -1, false);
    vector<int> for_do(n);
    for (int q = 0; q < n; q++) {
        for_do[q] = a[order[q]];
    }
    do_arr = DO(for_do);
}

int process_way(int x, int y, auto& func) {
    while (up[x] != up[y]) {
        if (height[up[x]] < height[up[y]]) {
            swap(x, y);
        }
        int z = up[x];
        func(tin[z], tin[x]+1);
        x = parents[z];
    }
    if (height[x] < height[y]) {
        swap(x, y);
    }
    func(tin[y], tin[x]+1);
    return y;
}
```

LCA_linear_memory

```
vector<vector<int>> d;
vector<int> parent, height, jump;

void make_LCA(int vertex, int p1, int h) {
    parent[vertex] = p1, height[vertex] = h;
    if (p1 != -1 && height[jump[p1]]-height[p1] == height[jump[jump[p1]]]-height[jump[p1]]) {
        jump[vertex] = jump[jump[p1]];
    } else {
        jump[vertex] = (p1 == -1 ? vertex : p1);
    }
    for (int q : d[vertex]) {
        if (q != p1) {
            make_LCA(q, vertex, h+1);
        }
    }
}

int k_ancestor(int vertex, int k) {
    int h = height[vertex]-k;
    while (height[vertex] > h) {
        vertex = (height[jump[vertex]] >= h ? jump[vertex] : parent[vertex]);
    }
    return vertex;
}

int LCA(int x, int y) {
    if (height[x] < height[y]) {
        swap(x, y);
    }
    x = k_ancestor(x, height[x]-height[y]);
    while (x != y) {
        if (jump[x] != jump[y]) {
            x = jump[x], y = jump[y];
        } else {
            x = parent[x], y = parent[y];
        }
    }
    return x;
}
```

Berlekamp

```
const int C = 1791179179;

int pow1(int x, int y) {
    if (y == 0) {
        return 1;
    }
    if (y % 2 == 0) {
        return pow1(x*x % C, y/2);
    }
    return pow1(x, y-1)*x % C;
}

vector<int> Berlekamp(vector<int> &rec) {
    int n = rec.size(), q1 = 0;
    while (q1 < n && rec[q1] == 0) {
        q1++;
    }
    if (q1 == n) {
        return {};
    }
```

```
        int t = rec[q1] % C, q2 = q1++;
        vector<int> was, now = vector<int>(q1, 0);
        for (; q1 < n; q1++) {
            int d = -rec[q1] % C;
            for (int q = 1; q <= now.size(); q++) {
                d = (d+now[q-1]*rec[q1-q]) % C;
            }
            if (d == 0) {
                continue;
            }
            vector<int> will = now;
            while (will.size() < q1-q2+(int)was.size()) {
                will.push_back(0);
            }
            int mul = d*pow1(t, C-2) % C;
            will[q1-q2-1] = (will[q1-q2-1]+mul) % C;
            for (int q = 0; q < was.size(); q++) {
                will[q1-q2+q] = (will[q1-q2+q]-was[q]*mul) % C;
            }
            was = now, now = will, t = d, q2 = q1;
        }
        for (int& q : now) {
            q = (q+C) % C, q -= (q > C/2)*C;
        }
        while (!now.empty() && now.back() == 0) {
            now.pop_back();
        }
        return now;
    }

    int stupid(int n) {
        //to do
    }

    int find_n(int n, bool flag = false) {
        int k = 57;
        vector<int> a = {0};
        for (int q = 1; q < (flag ? n+1 : k); q++) {
            a.push_back(stupid(q));
        }
        vector<int> rec = Berlekamp(a);
        if (flag) {
            for (int q : rec) {
                cout << q << ' ';
            }
            cout << endl;
        }
        for (int q = k; q <= n; q++) {
            a.push_back(0);
            for (int q1 = 1; q1 <= rec.size(); q1++) {
                a.back() += a[q-q1]*rec[q1-1] % C;
            }
            a.back() = (a.back() % C+C) % C;
        }
        return a[n];
    }
```

FFT_complex

```
    const ld PI = numbers::pi_v<ld>;
    using Complex = complex<ld>;

    int reverse_bits(int x, int log1) {
        int y = 0;
        for (int q = 0; q < log1; q++) {
            y |= (((x >> q) & 1) << (log1-q-1));
        }
        return y;
    }

    vector<int> rev_bits;

    void build_rev_bits(int log1) {
        int n = (1 << log1);
        rev_bits.assign(n, -1);
        for (int q = 0; q < n; q++) {
            int q1 = reverse_bits(q, log1);
            rev_bits[q] = min(q, q1);
        }
    }

    void FFT(vector<Complex>& a, int log1) {
        int n = (1 << log1);
        for (int q = 0; q < n; q++) {
            swap(a[q], a[rev_bits[q]]);
        }
        for (int q = 1; q < n; q <<= 1) {
            Complex root(cosl(PI/q), sinl(PI/q));
            for (int q1 = 0; q1 < n; q1 += (q << 1)) {
                Complex now = 1;
                for (int q2 = q1; q2 < q1+q; q2++) {
                    Complex x = a[q2], y = a[q2+q]*now;
                    a[q2] = x+y, a[q2+q] = x-y;
                    now *= root;
                }
            }
        }
    }

    void IFFT(vector<Complex>& a, int log1) {
        int n = (1 << log1);
        FFT(a, log1);
        reverse(a.begin()+1, a.end());
        for (Complex& q : a) {
            q /= n;
        }
    }

    int get_degree(int n) {
        int log1 = 0;
        while ((1 << log1) < n) {
            log1++;
        }
        return log1;
    }

    vector<Complex> multiply(vector<Complex> a, vector<Complex> b) {
        int len = (int)a.size()+(int)b.size()-1;
        int log1 = get_degree(len), n = (1 << log1);
        build_rev_bits(log1);
        a.resize(n, 0), b.resize(n, 0);
        FFT(a, log1), FFT(b, log1);
        for (int q = 0; q < n; q++) {
            a[q] = a[q]*b[q];
        }
        IFFT(a, log1);
        return {a.begin(), a.begin()+len};
    }
```

FFT_divide

```
vector<int> operator*(vector<int> a, int x) {
    x = (x % C+C) % C;
    for (int& q : a) {
        q *= x, q %= C;
    }
    return a;
}

vector<int> inverse(vector<int>& A, int n) {
    assert(A[0] != 0);
    vector<int> B0 = {pow1(A[0], C-2)};
    int N = A.size();
    while (B0.size() <= n) {
        int k = B0.size(), len = min(2*k, N);
        vector<int> A_k(A.begin(), A.begin()+len);
        auto A_B0 = multiply(A_k, B0);
        A_B0.erase(A_B0.begin(), A_B0.begin()+k);
        if (A_B0.empty()) {
            break;
        }
        auto B1 = multiply(A_B0*(-1), B0);
        B0.insert(B0.end(), B1.begin(), B1.begin()+k);
    }
    B0.resize(n+1, 0);
    return B0;
}

vector<int> divide(vector<int> A, vector<int> B) {
    int n = A.size(), m = B.size();
    if (n < m) {
        return {0};
    }
    int k = n-m+1;
    reverse(A.begin(), A.end());
    A.resize(k);
    reverse(B.begin(), B.end());
    auto Q = multiply(A, inverse(B, k));
    Q.resize(k, 0);
    reverse(Q.begin(), Q.end());
    return Q;
}

vector<int> remainder(vector<int> A, vector<int> B) {
    int n = A.size(), m = B.size();
    if (n < m) {
        return A;
    }
    auto Q = divide(A, B);
    Q.resize(min(m, n-m+1));
    auto QB = multiply(Q, B);
    for (int q = 0; q < m; q++) {
        A[q] -= QB[q];
        A[q] += C*(A[q] < 0);
    }
    int ind = m-1;
    while (ind > 0 && A[ind] == 0) {
        ind--;
    }
    return {A.begin(), A.begin()+ind+1};
}

vector<int> pow1(vector<int> A, int y, vector<int>& MOD) {
    if (y == 0) {
        return {1};
    }
    if (y % 2 == 0) {
        auto res = remainder(multiply(A, A), MOD);
        return pow1(res, y/2, MOD);
    }
    auto res = multiply(pow1(A, y-1, MOD), A);
    return remainder(res, MOD);
}

int get_rec_coef(vector<int>& rec, vector<int>& a, int N) {
    vector<int> Q = rec*(-1);
    Q.push_back(1);
    auto coefs = pow1({0, 1}, N, Q);
    int ans = 0;
    for (int q = 0; q < coefs.size(); q++) {
        ans += coefs[q]*a[q] % C;
    }
    return ans % C;
}
```

FFT_modulo

```
const int g = 31;
vector<int> rev_bits;

int reverse_bits(int x, int log1) {
    int y = 0;
    for (int q = 0; q < log1; q++) {
        y |= (((x >> q) & 1) << (log1-q-1));
    }
    return y;
}

void build_rev_bits(int log1) {
    int n = (1 << log1);
    rev_bits.assign(n, -1);
    for (int q = 0; q < n; q++) {
        int q1 = reverse_bits(q, log1);
        rev_bits[q] = min(q, q1);
    }
}

void FFT(int* a, int log1) {
    int n = (1 << log1);
    for (int q = 0; q < n; q++) {
        swap(a[q], a[rev_bits[q]]);
    }
    for (int q = 1; q < n; q <<= 1) {
        int root = pow1(g, (C-1)/(q << 1));
        for (int q1 = 0; q1 < n; q1 += (q << 1)) {
            int now = 1;
            for (int q2 = q1; q2 < q1+q; q2++) {
                int x = a[q2], y = a[q2+q]*now % C;
                a[q2] = x+y-(x+y >= C)*C;
                a[q2+q] = x-y+(x-y < 0)*C;
                now = now*root % C;
            }
        }
    }
}

void IFFT(int* a, int log1) {
        int n = (1 << log1);
        FFT(a, log1);
        reverse(a+1, a+n);
        int rev_n = pow1(n, C-2);
        for (int q = 0; q < n; q++) {
            a[q] *= rev_n, a[q] %= C;
        }
    }

    int get_degree(int n) {
        int log1 = 0;
        while ((1 << log1) < n) {
            log1++;
        }
        return log1;
    }

    vector<int> multiply(vector<int>& aa, vector<int>& bb) {
        int len = (int)aa.size()+(int)bb.size()-1;
        int log1 = get_degree(len), n = (1 << log1);
        build_rev_bits(log1);
        int a[n], b[n];
        fill(a, a+n, 0), fill(b, b+n, 0);
        copy(aa.begin(), aa.end(), a);
        copy(bb.begin(), bb.end(), b);
        FFT(a, log1), FFT(b, log1);
        for (int q = 0; q < n; q++) {
            a[q] = a[q]*b[q] % C;
        }
        IFFT(a, log1);
        vector<int> ans(a, a+len);
        return ans;
    }
```

FPS

```
    vector<int> Newton_method(vector<int> f0, auto& P, auto& dP, int n) {
        while (f0.size() <= n) {
            int k = f0.size();
            auto num = P(f0), den = dP(f0);
            den = inverse(den, 2*k);
            auto f1 = multiply(num*(-1), den);
            for (int q = 0; q < k; q++) {
                f1[q] += f0[q];
                f1[q] -= C*(f1[q] >= C);
            }
            f0 = {f1.begin(), f1.begin()+2*k};
        }
        return {f0.begin(), f0.begin()+n+1};
    }

    vector<int> square_root(vector<int>& A, int n) {
        assert(A[0] == 1);
        auto P = [&A](vector<int>& f) {
            auto f_2 = multiply(f, f);
            int len = max(A.size(), f_2.size());
            f_2.resize(len, 0);
            for (int q = 0; q < A.size(); q++) {
                f_2[q] -= A[q];
                f_2[q] += C*(f_2[q] < 0);
            }
            return f_2;
        };
        auto dP = [](vector<int>& f) {
```

```
        return f*2;
    };
    vector<int> f0 = {1};
    return Newton_method(f0, P, dP, n);
}
```

online_FFT

```
vector<int> online_FFT(int a0, vector<int> b, int c0, auto& func) {
    int n = (int)b.size();
    int log1 = get_degree(n), len = (1 << log1);
    b.resize(len, 0);
    vector<vector<int>> blocks(log1);
    for (int q = 0; q < log1; q++) {
        for (int q1 = (1 << q); q1 < (1 << (q+1)); q1++) {
            blocks[q].push_back(b[q1]);
        }
    }
    vector<int> a(2*len, 0), c(2*len, 0);
    a[0] = a0, c[0] = c0;
    for (int q = 1; q < n; q++) {
        int deg = 1, ind = 0;
        while (q % deg == 0) {
            vector<int> a_i(a.begin()+q-deg, a.begin()+q);
            vector<int> now = multiply(a_i, blocks[ind]);
            for (int q1 = 0; q1 < now.size(); q1++) {
                c[q+q1] += now[q1], c[q+q1] %= C;
            }
            deg *= 2, ind++;
        }
        func(q, a, c);
    }
    return {c.begin(), c.begin()+n};
}
```

Floor_sum

```
int floor_sum(int a, int b, int k) { // [a/b]+...+[k*a/b]
    if (a >= b) {
        int t = a/b, r = a % b;
        return k*(k+1)/2*t+floor_sum(r, b, k);
    }
    if (a == 0) {
        return 0;
    }
    int m = k*a/b, total = k*m;
    int complement = floor_sum(b, a, m);
    int on_diag = k*gcd(a, b)/b;
    return total+on_diag-complement;
}

int floor_sum(int a, int b, int l, int r) {
    return floor_sum(a, b, max(0LL, r-1))-floor_sum(a, b, max(0LL, l-1));
}

int sum_mod(int a, int b, int l, int r) {
    int amount = r*(r-1)/2-l*(l-1)/2;
    return a*amount-b*floor_sum(a, b, l, r);
}
```

ax_by_c

```
p build_gcd_coefs(int a, int b) {
```

```
|      if (b == 0) {
|        return {1, 0};
|      }
|      int m = a/b;
|      auto [y, x] = build_gcd_coefs(b, a-m*b);
|      return {x, y-m*x};
|    }
|
|    p gcd_coef(int a, int b) {
|      int mx = max(a, b), mn = min(a, b);
|      auto [x, y] = build_gcd_coefs(mx, mn);
|      if (a < b) {
|        swap(x, y);
|      }
|      return {x, y};
|    }
|
|    p ax_by_c(int a, int b, int c) {
|      auto [x, y] = gcd_coef(a, b);
|      int g = a*x+b*y;
|      if (c % g != 0) {
|        return {-1, -1};
|      }
|      a /= g, b /= g, c /= g;
|      x *= c, y *= c;
|      int t = x/b;
|      x -= b*t, y += a*t;
|      if (x < 0) {
|        x += b, y -= a;
|      }
|      return {x, y};
|    }
```

C_k_n_any_modulo

```
|    int pow1(int x, int y, int C) {
|      if (y == 0) {
|        return 1;
|      }
|      if (y % 2 == 0) {
|        return pow1(x*x % C, y/2, C);
|      }
|      return pow1(x, y-1, C)*x % C;
|    }
|
|    int phi(int n, const vector<int> primes) {
|      int ans = n;
|      for (int q : primes) {
|        ans -= ans/q;
|      }
|      return ans;
|    }
|
|    vector<int> fact, rev_fact;
|
|    void make_fact(int n, int C, vector<int> primes) {
|      fact = {1};
|      vector<int> numbers = {0};
|      for (int q = 1; q <= n; q++) {
|        int now = q;
|        for (int q1 : primes) {
|          while (now % q1 == 0) {
|            now /= q1;
|          }
```

```
        }
        fact.push_back(fact.back()*now % C);
        numbers.push_back(now);
    }
    rev_fact = {pow1(fact.back(), phi(C, primes)-1, C)};
    for (int q = min(n, C-1); q > 0; q--) {
        rev_fact.push_back(rev_fact.back()*numbers[q] % C);
    }
    reverse(rev_fact.begin(), rev_fact.end());
}

int p_degree(int n, int C) {
    int ans = 0, deg = C;
    while (deg <= n) {
        ans += n/deg, deg *= C;
    }
    return ans;
}

int C_k_n(int k, int n, int C, vector<int> primes) {
    int ans = fact[n]*rev_fact[k] % C*rev_fact[n-k] % C;
    for (int q : primes) {
        int deg = p_degree(n, q)-p_degree(k, q)-p_degree(n-k, q);
        ans = ans*pow1(q, deg, C) % C;
    }
    return ans;
}

vector<int> factor(int n) {
    vector<int> primes;
    int sqrt1 = sqrt(n);
    for (int q = 2; q <= sqrt1; q++) {
        if (n % q == 0) {
            primes.push_back(q);
        }
        while (n % q == 0) {
            n /= q;
        }
    }
    if (n > 1) {
        primes.push_back(n);
    }
    return primes;
}
```

C_k_n_modulo_p

```
int pow1(int x, int y, int C) {
    if (y == 0) {
        return 1;
    }
    if (y % 2 == 0) {
        return pow1(x*x % C, y/2, C);
    }
    return pow1(x, y-1, C)*x % C;
}

vector<int> fact, rev_fact;

void make_fact(int C) {
    fact = {1};
    for (int q = 1; q < C; q++) {
        fact.push_back(fact.back()*q % C);
    }
```

```
        rev_fact = {pow1(fact.back(), C-2, C)};
        for (int q = C-1; q > 0; q--) {
            rev_fact.push_back(rev_fact.back()*q % C);
        }
        reverse(rev_fact.begin(), rev_fact.end());
    }

    int get_fact(int n, int C) {
        if (n < C) {
            return fact[n];
        }
        int ans = fact[n % C]*pow1(fact.back(), n/C, C) % C;
        return ans*get_fact(n/C, C) % C;
    }

    int get_rev_fact(int n, int C) {
        if (n < C) {
            return rev_fact[n];
        }
        int ans = rev_fact[n % C]*pow1(rev_fact.back(), n/C, C) % C;
        return ans*get_rev_fact(n/C, C) % C;
    }

    int p_degree(int n, int C) {
        int ans = 0, deg = C;
        while (deg <= n) {
            ans += n/deg, deg *= C;
        }
        return ans;
    }

    int C_k_n(int k, int n, int C) {
        if (k < 0 || k > n) {
            return 0;
        }
        if (p_degree(n, C) > p_degree(k, C)+p_degree(n-k, C)) {
            return 0;
        }
        return get_fact(n, C)*get_rev_fact(k, C) % C*get_rev_fact(n-k, C) % C;
    }
```

N_th_root

```
    int pow1(int x, int y, int C) {
        if (y == 0) {
            return 1;
        }
        if (y % 2 == 0) {
            return pow1(x*x % C, y/2, C);
        }
        return pow1(x, y-1, C)*x % C;
    }

    vector<int> factor(int n) {
        vector<int> primes;
        int sqrt1 = sqrt(n);
        for (int q = 2; q <= sqrt1; q++) {
            if (n % q == 0) {
                primes.push_back(q);
            }
            while (n % q == 0) {
                n /= q;
            }
        }
        if (n > 1) {
            primes.push_back(n);
        }
        return primes;
    }

    int find_g(int C) {
        vector<int> primes = factor(C-1);
        for (int q = 1;; q++) {
            bool flag = true;
            for (int q1 : primes) {
                if (pow1(q, (C-1)/q1, C) == 1) {
                    flag = false;
                    break;
                }
            }
            if (flag) {
                return q;
            }
        }
    }

    int g_degree(int x, int g, int C) {
        int sqrt1 = sqrt(C);
        unordered_map<int, int> a;
        a.reserve(sqrt1);
        int now = 1;
        for (int q = 0; q < sqrt1; q++) {
            a[now] = q;
            now = now*g % C;
        }
        int rev_sqrt = pow1(now, C-2, C);
        for (int q = 0; q <= C/sqrt1; q++) {
            if (a.find(x) != a.end()) {
                return q*sqrt1+a[x];
            }
            x = x*rev_sqrt % C;
        }
        return -1;
    }

    int phi(int n) {
        vector<int> fact = factor(n);
        int ans = n;
        for (int q : fact) {
            ans -= ans/q;
        }
        return ans;
    }

    p ax_by_c(int a, int b, int c) {
        int t = __gcd(a, b);
        if (c % t != 0) {
            return {-1, -1};
        }
        a /= t, b /= t, c /= t;
        int x = (c*pow1(a, phi(b)-1, b) % b+b) % b;
        int y = (c-a*x)/b;
        return {x, y};
    }

    int sqrt_b(int a, int b, int C) {
        if (a == 0) {
            return 0;
    }
    int g = find_g(C);
    int deg_a = g_degree(a, g, C);
    int x = ax_by_c(b, C-1, deg_a).first;
    return (x == -1 ? -1 : pow1(g, x, C));
}
```

Square_root

```
int pow1(int x, int y, int C) {
    if (y == 0) {
        return 1;
    }
    if (y % 2 == 0) {
        return pow1(x*x % C, y/2, C);
    }
    return pow1(x, y-1, C)*x % C;
}

mt19937 randint(17957179);

struct Poly {
    int a, b;
};

Poly mul(Poly x, Poly y, int A, int C) {
    int a = x.a*y.b+x.b*y.a;
    int b = x.b*y.b+x.a*y.a % C*A;
    return {a % C, b % C};
}

int find_sqrt(int a, int C) {
    while (true) {
        int t = randint() % (C-1)+1;
        Poly x_t(1, t);
        int deg = (C-1)/2;
        Poly poly(0, 1);
        while (deg > 0) {
            if (deg & 1) {
                poly = mul(poly, x_t, a, C);
            }
            x_t = mul(x_t, x_t, a, C);
            deg >>= 1;
        }
        auto [A, B] = poly;
        B = (B+C-1) % C;
        int val = A*A % C*a % C-B*B % C;
        if (A == 0 || val != 0) {
            continue;
        }
        int rev_a = pow1(A, C-2, C);
        return B*rev_a % C;
    }
}

int square_root(int a, int C) {
    if (a == 0 || C == 2) {
        return a;
    }
    int sign = pow1(a, (C-1)/2, C);
    if (sign != 1) {
        return -1;
    }
    if (C % 4 == 3) {
        return pow1(a, (C+1)/4, C);
    }
    return find_sqrt(a, C);
}
```

basic

```
const ld E = 1e-8;

template <typename T>
struct NumT {
    T val;

    NumT(T val): val(val) {}

    operator T() {
        return val;
    }

    bool operator==(const NumT& other) const {
        return abs(val-other.val) < E;
    }
};

using Num = NumT<ld>;

auto operator<=>(const Num& x, const Num& y) {
    if (x == y) {
        return x.val <=> x.val;
    }
    return x.val <=> y.val;
}

istream& operator>>(istream& in, Num& x) {
    int val;
    cin >> val;
    x.val = val;
    return in;
}

int sign(Num x) {
    return (x > Num(0))-(x < Num(0));
}

struct Pt {
    Num x = 0, y = 0;

    bool operator==(const Pt& other) const {
        return x == other.x && y == other.y;
    }

    Pt operator+(Pt other) {
        return {x+other.x, y+other.y};
    }

    Pt operator-(Pt other) {
        return {x-other.x, y-other.y};
    }

    Pt operator*(Num t) {
        return {x*t, y*t};
    }
};

```

```
    Num dot(Pt x, Pt y) {
        return x.x*y.x+x.y*y.y;
    }

    Num cross(Pt x, Pt y) {
        return x.x*y.y-x.y*y.x;
    }

    Num dist_2(Pt x, Pt y) {
        return dot(y-x, y-x);
    }

    ld angle(Pt x) {
        ld alpha = atan2l(x.y, x.x);
        ld pi = numbers::pi_v<ld>;
        return alpha+2*pi*(alpha < -E);
    }

    struct Line {
        Pt a, dir;

        int side(Pt x) {
            return sign(cross(dir, x-a));
        }

        Pt proj(Pt x) {
            ld t = dot(x-a, dir)/dot(dir, dir);
            return a+dir*t;
        }
    };

    Line make_line(Pt x, Pt y) {
    //    ld t = 1/sqrtl(dist_2(x, y));
        int t = 1;
        return {x, (y-x)*t};
    }

    Pt inter(Line line1, Line line2) {
        Num det = cross(line1.dir, line2.dir);
        if (det == Num(0)) {
            return {NAN, NAN};
        }
        Pt r = line2.a-line1.a;
        Num det_x = cross(r, line2.dir);
        ld t = det_x/det;
        return line1.a+line1.dir*t;
    }

    bool on_seg(Pt a, Pt b, Pt x) {
        return cross(x-a, x-b) == Num(0) && dot(x-a, x-b) <= Num(0);
    }

    struct Cir {
        Pt c;
        Num r = 0;

        int in(Pt x) {
            return sign(r*r-dist_2(c, x));
        }
    };

    vector<Pt> inter(Cir cir, Line line) {
        Pt proj = line.proj(cir.c);
        Num d_2 = dist_2(proj, cir.c);
        Num r_2 = cir.r*cir.r;
        if (d_2 > r_2) {
            return {};
        }
        if (d_2 == r_2) {
            return {proj};
        }
        Pt dir = line.dir;
        ld t = sqrtl((r_2-d_2)/dot(dir, dir));
        return {proj-dir*t, proj+dir*t};
    }

    Line rad_axis(Cir cir1, Cir cir2) {
        Num d_2 = dist_2(cir1.c, cir2.c);
        Num r_2 = cir1.r*cir1.r-cir2.r*cir2.r;
        ld t = (d_2+r_2)/(2*d_2);
        Pt dir = cir2.c-cir1.c;
        return {cir1.c+dir*t, {-dir.y, dir.x}};
    }

    vector<Pt> inter(Cir cir1, Cir cir2) {
        if (cir1.c == cir2.c) {
            return {};
        }
        return inter(cir1, rad_axis(cir1, cir2));
    }

    vector<Pt> tangents(Cir cir, Pt x) {
        Num r_2 = dist_2(cir.c, x)-cir.r*cir.r;
        ld r = sqrtl(max<ld>(0, r_2));
        return inter({x, r}, cir);
    }
```

polygons

```
    struct Polygon {
        vector<Pt> a;
        int n;

        Polygon(const vector<Pt>& a_): a(a_) {
            auto w = unique(a.begin(), a.end());
            n = w-a.begin();
            if (n == 0) {
                return;
            }
            n -= (n > 1 && a[0] == a[n-1]);
            a.resize(n);
            a.push_back(a[0]);
        }
    };

    int belong_convex(Polygon& a, Pt x) {
        if (a.n <= 1) {
            return (a.n == 1 && a.a[0] == x)-1;
        }
        Pt pt = a.a[0];
        if (cross(a.a[1]-pt, x-pt) <= Num(0)) {
            return on_seg(a.a[0], a.a[1], x)-1;
        }
        if (cross(a.a[a.n-1]-pt, x-pt) >= Num(0)) {
            return on_seg(a.a[0], a.a[a.n-1], x)-1;
        }
        int l = 0, r = a.n;
        while (r-l > 1) {
            int m = (l+r)/2;
            if (cross(a.a[m]-pt, x-pt) > Num(0)) {
                l = m;
            } else {
                r = m;
            }
        }
        return make_line(a.a[l], a.a[r]).side(x);
    }

    bool need_pop_back(vector<Pt>& a, Pt x) {
        int m = (int)a.size();
        return m >= 2 && cross(a[m-1]-a[m-2], x-a[m-2]) <= Num(0);
    }

    vector<Pt> build_envelope(vector<Pt>& a) {
        int n = (int)a.size();
        vector<Pt> ans = {a[0]};
        for (int q = 1; q < n; q++) {
            while (need_pop_back(ans, a[q])) {
                ans.pop_back();
            }
            ans.push_back(a[q]);
        }
        return ans;
    }

    Polygon convex_hull(vector<Pt> a) {
        if (a.empty()) {
            return {{}};
        }
        ranges::sort(a, {}, [](Pt x) {return pair{x.x, x.y};});
        vector<Pt> up = {a[0]}, down = {a[0]};
        for (Pt pt : a) {
            Num c = cross(a.back()-a[0], pt-a[0]);
            if (c > Num(0)) {
                up.push_back(pt);
            } else if (c < Num(0)) {
                down.push_back(pt);
            }
        }
        up.push_back(a.back());
        down.push_back(a.back());
        reverse(up.begin(), up.end());
        up = build_envelope(up);
        down = build_envelope(down);
        down.insert(down.end(), up.begin()+1, up.end()-1);
        return {down};
    }

    bool add_line(deque<Line>& hull, Line line) {
        int len = (int)hull.size();
        while (len > 1 && line.side(inter(hull[len-1], hull[len-2])) == -1) {
            hull.pop_back();
            len--;
        }
        while (len > 1 && line.side(inter(hull[0], hull[1])) == -1) {
            hull.pop_front();
            len--;
        }
        if (len > 0 && cross(line.dir, hull.back().dir) == Num(0)) {
            if (dot(line.dir, hull.back().dir) < Num(0)) {
                return false;
            }
        if (cross(hull.back().dir, line.a-hull.back().a) > Num(0)) {
            hull.pop_back();
        } else {
            return true;
        }
    }
    if (len == 1 && cross(hull[0].dir, line.dir) < Num(0)) {
        return false;
    }
    hull.push_back(line);
    return true;
}

vector<Line> inter_lines(vector<Line>& a) {
    deque<Line> hull;
    for (Line line : a) {
        if (!add_line(hull, line)) {
            return {};
        }
    }
    if (hull.empty() || !add_line(hull, hull[0])) {
        return {};
    }
    return {hull.begin(), hull.end()-1};
}

Polygon half_planes_inter(vector<Line> A) {
    A.emplace_back(Pt(-INF, -INF), Pt(1, 0));
    A.emplace_back(Pt(INF, -INF), Pt(0, 1));
    A.emplace_back(Pt(INF, INF), Pt(-1, 0));
    A.emplace_back(Pt(-INF, INF), Pt(0, -1));

    vector<pair<Line, ld>> aa;
    for (Line line : A) {
        aa.emplace_back(line, angle(line.dir));
    }
    ranges::sort(aa, {}, [](auto x) {return x.second;});
    vector<Line> a;
    for (auto q : aa) {
        a.push_back(q.first);
    }
    vector<Line> hull = inter_lines(a);

    int n = (int)hull.size();
    vector<Pt> ans;
    for (int q = 0; q < n; q++) {
        ans.push_back(inter(hull[q], hull[(q+1) % n]));
    }
    return {ans};
}
```

Grice_Misra_sieve

```
vector<int> Grice_Misra_sieve(int n) {
    vector<int> primes, lcp(n+1, 0);
    for (int q = 2; q <= n; q++) {
        if (lcp[q] == 0) {
            lcp[q] = q;
            primes.push_back(q);
        }
        for (int q1 = 0; q1 < primes.size() && primes[q1] <= lcp[q] && q*primes[q1] <= n; q1++) {
            lcp[q*primes[q1]] = primes[q1];
        }
    }
```

```
        return lcp;
    }
```

Binary_Gauss

```
    #include <tr2/dynamic_bitset>
    using Bitset = tr2::dynamic_bitset<>;

    struct Gauss {
        vector<Bitset> a;
        int n, m;

        explicit Gauss(auto& a_): n(a_.size()), m(a_[0].size()) {
            for (int q = 0; q < n; q++) {
                a.emplace_back(m);
                for (int q1 = 0; q1 < m; q1++) {
                    a[q][q1] = a_[q][q1];
                }
            }
        }

        void subtract(int q, int q1) {
            for (int q2 = 0; q2 < n; q2++) {
                if (q != q2 && a[q2][q1]) {
                    a[q2] ^= a[q];
                }
            }
        }

        void gauss() {
            for (int q = 0; q < n; q++) {
                int q1 = (int)a[q].find_first();
                if (q1 != m) {
                    subtract(q, q1);
                }
            }
        }

        auto annihilator() const {
            vector<int> leader(n);
            for (int q = 0; q < n; q++) {
                leader[q] = (int)a[q].find_first();
            }
            vector<Bitset> ans;
            for (int q1 = 0; q1 < m; q1++) {
                if (ranges::find(leader, q1) != leader.end()) {
                    continue;
                }
                ans.emplace_back(m+1);
                for (int q = 0; q < n; q++) {
                    ans.back()[leader[q]] = a[q][q1];
                }
                ans.back().resize(m);
                ans.back()[q1] = true;
            }
            return ans;
        }
    };
```

SLAE_solve

```
    const ld E = 1e-8;

```

```
struct Matrix {
    vector<vector<ld>> a;
    int n, m;

    Matrix(auto& a_): n(a_.size()), m(a_[0].size()) {
        a.assign(n, vector<ld>(m));
        for (int q = 0; q < n; q++) {
            for (int q1 = 0; q1 < m; q1++) {
                a[q][q1] = a_[q][q1];
            }
        }
    }

    Matrix(Matrix&) = default;

    int find_leader(int q) {
        int q1 = 0;
        while (q1 < m && abs(a[q][q1]) < E) {
            q1++;
        }
        return q1;
    }

    void subtract(int q, int q1) {
        ld val = a[q][q1];
        for (int q2 = 0; q2 < n; q2++) {
            if (q == q2 || abs(a[q2][q1]) < E) {
                continue;
            }
            ld coef = a[q2][q1]/val;
            for (int q3 = 0; q3 < m; q3++) {
                a[q2][q3] -= a[q][q3]*coef;
            }
        }
    }

    void gauss() {
        for (int q = 0; q < n; q++) {
            int q1 = find_leader(q);
            if (q1 != m) {
                subtract(q, q1);
            }
        }
    }
};

vector<ld> solve_SLAE(Matrix a, vector<ld>& s) {
    a.m++;
    for (int q = 0; q < a.n; q++) {
        a.a[q].push_back(s[q]);
    }
    a.gauss();
    vector<ld> ans(a.m-1, 0);
    for (int q = 0; q < a.n; q++) {
        int q1 = a.find_leader(q);
        if (q1 == a.m-1) {
            return {};
        }
        if (q1 != a.m) {
            ld val = a.a[q][q1];
            ld need = a.a[q].back();
            ans[q1] = need/val;
        }
    }
    return ans;
}
```

Or_And_convolution

```
template <bool Rev = false>
vector<int> SOS(vector<int> a) {
    int n = (int)a.size();
    for (int q1 = 1; q1 < n; q1 <<= 1) {
        for (int q2 = q1; q2 < n; q2 += (q1 << 1)) {
            for (int q = q2; q < q2+q1; q++) {
                if constexpr (!Rev) {
                    a[q] += a[q ^ q1];
                } else {
                    a[q] -= a[q ^ q1];
                }
            }
        }
    }
    for (int& q : a) {
        q = (q % C+C) % C;
    }
    return a;
}

vector<int> num_ones;

void make_num_ones(int n) {
    num_ones.assign(1 << n, 0);
    for (int q = 0; q < n; q++) {
        num_ones[1 << q] = 1;
    }
    for (int q = 1; q < (1 << n); q++) {
        int last_bit = q-(q & (q-1));
        num_ones[q] = num_ones[q ^ last_bit]+num_ones[last_bit];
    }
}

int SOS_value(int q, const vector<int> &a) {
    int ans = (1-((num_ones[q] & 1) << 1))*a[0];
    for (int q1 = q; q1 > 0; q1 = ((q1-1) & q)) {
        ans += (1-((num_ones[q ^ q1] & 1) << 1))*a[q1];
    }
    return (ans % C+C) % C;
}

auto or_convolution_SOS(const vector<int> &a, const vector<int> &b) {
    int n = (int)a.size();
    vector<int> c(n);
    for (int q = 0; q < n; q++) {
        c[q] = a[q]*b[q] % C;
    }
    return c;
}

vector<int> or_convolution(const vector<int>& a, const vector<int>& b) {
    return SOS<true>(or_convolution_SOS(SOS(a), SOS(b)));
}

vector<int> and_convolution(vector<int> a, vector<int> b) {
    int n = (int)a.size(), ALL = n-1;
    for (int q = 0; q < (n >> 1); q++) {
        swap(a[q], a[q ^ ALL]);
        swap(b[q], b[q ^ ALL]);
    }
    vector<int> c = or_convolution(a, b);
    for (int q = 0; q < (n >> 1); q++) {
        swap(c[q], c[q ^ ALL]);
    }
    return c;
}
```

Pollard

```
__int128 pow2(__int128 x, __int128 y, __int128 C) {
    if (y == 0) {
        return 1;
    }
    if (y % 2 == 0) {
        return pow2(x*x % C, y/2, C);
    }
    return pow2(x, y-1, C)*x % C;
}

bool Miller_Rabin_test(int a, int n) {
    int d = n-1;
    while ((d & 1) ^ 1) {
        d >>= 1;
    }
    __int128 now = pow2(a, d, n);
    if (now == 1) {
        return true;
    }
    while (d < n-1) {
        if (now == n-1) {
            return true;
        }
        now = now*now % n, d <<= 1;
    }
    return false;
}

mt19937 randint(17957179);

bool Miller_Rabin(int n, int k = 20) {
    if (n == 1) {
        return false;
    }
    for (int q = 0; q < k; q++) {
        if (!Miller_Rabin_test(randint() % (n-1)+1, n)) {
            return false;
        }
    }
    return true;
}

int f_Pollard(__int128 x, int n) {
    return (x*x+3) % n;
}

vector<int> make_Pollard(int n) {
    if (Miller_Rabin(n)) {
        return {n};
    }
    int x = randint() % (n-1)+1;
    int y = f_Pollard(x, n);
    while (__gcd(n, abs(y-x)) == 1) {
        x = f_Pollard(x, n);
```

```
            y = f_Pollard(f_Pollard(y, n), n);
        }
        if (x == y) {
            return make_Pollard(n);
        }
        int d = __gcd(n, abs(y-x));
        vector<int> ans = make_Pollard(d);
        for (int q : make_Pollard(n/d)) {
            ans.push_back(q);
        }
        return ans;
    }

    vector<int> Pollard(int n) {
        vector<int> primes, small = {2, 3, 5, 7};
        for (int q : small) {
            while (n % q == 0) {
                primes.push_back(q);
                n /= q;
            }
        }
        if (n == 1) {
            return primes;
        }
        multiset<int> all;
        for (int q : make_Pollard(n)) {
            all.insert(q);
        }
        for (int q : all) {
            primes.push_back(q);
        }
        return primes;
    }
```

Smiths_theory

```
    vector<vector<int>> get_graph() {
        int n, m;
        cin >> n >> m;
        vector<vector<int>> d(n);
        for (int q = 0; q < m; q++) {
            int x, y;
            cin >> x >> y;
            x--, y--;
            d[x].push_back(y);
        }
        return d;
    }

    vector<int> get_smith(vector<vector<int>> &d) {
        int n = (int)d.size();
        vector<vector<int>> d1(n);
        for (int q = 0; q < n; q++) {
            for (int q1 : d[q]) {
                d1[q1].push_back(q);
            }
        }
        vector<int> smith(n, -1), all(n);
        iota(all.begin(), all.end(), 0);
        bool continue1 = true;
        for (int nim = 0; continue1; nim++) {
            continue1 = false;
            vector<int> num(n, 0);
            for (int q : all) {
```

```
            for (int q1 : d[q]) {
                num[q] += (smith[q1] == -1);
            }
        }
        vector<p> is;
        vector<bool> is_move(n, false), is_now(n, false);
        for (int q : all) {
            is.emplace_back(num[q], q);
            is_now[q] = true;
        }
        sort(is.rbegin(), is.rend());
        vector<int> ind(n);
        for (int q = 0; q < is.size(); q++) {
            ind[is[q].second] = q;
        }
        all = {};
        while (!is.empty() && is.back().first == 0) {
            continue1 = true;
            int vertex = is.back().second;
            is.pop_back();
            if (!is_now[vertex]) {
                continue;
            }
            is_now[vertex] = false, smith[vertex] = nim;
            for (int q : d1[vertex]) {
                if (is_move[q] || smith[q] != -1) {
                    continue;
                }
                if (is_now[q]) {
                    all.push_back(q);
                    is_now[q] = false;
                }
                is_move[q] = true;
                for (int q1 : d1[q]) {
                    if (!is_now[q1]) {
                        continue;
                    }
                    auto w = lower_bound(is.rbegin(), is.rend(), p(num[q1], -INF));
                    auto w1 = is.rbegin()+(int)is.size()-ind[q1]-1;
                    w1->first--, num[q1]--;
                    swap(*w, *w1);
                    swap(ind[w->second], ind[w1->second]);
                }
            }
        }
    }
    return smith;
}

int sum_games_result(vector<vector<int>> &d1, vector<vector<int>> &d2,
                     vector<int> &smith1, vector<int> &smith2, int x, int y) {
    if (smith1[x] == -1 && smith2[y] == -1) {
        return -1;
    }
    if (smith1[x] != -1 && smith2[y] != -1) {
        return (smith1[x] ^ smith2[y]) == 0;
    }
    bool was_swap = false;
    if (smith1[x] != -1) {
        swap(d1, d2);
        swap(smith1, smith2);
        was_swap = true;
        swap(x, y);
    }
```

```
        bool flag = false;
        for (int q : d1[x]) {
            flag |= (smith1[q] == smith2[y]);
        }
        if (was_swap) {
            swap(d1, d2);
            swap(smith1, smith2);
        }
        return flag-1;
    }
```

Annealing_simulation

```
    mt19937 randint(179);
    const int MN = (1LL << 20);

    bool P(int x, int y, ld t) {
        ld is = exp((y-x)/t);
        return randint() % MN < is*MN;
    }

    int get_score(int x, vector<int>& a) {
        int ans = 0;
        for (int q : a) {
            ans += min(3LL, num_bits[q ^ x]);
        }
        return ans;
    }

    int delta_score(int ind, int x, vector<int> &a) {
        int score = 0, w = a[ind];
        a[ind] = x;
        score += get_score(a[ind], a);
        a[ind] = w;
        score -= get_score(a[ind], a);
        return score*2;
    }

    auto annealing(int n) {
        vector<int> a(1 << k);
        iota(a.begin(), a.end(), 0);
        shuffle(a.begin(), a.end(), randint);
        a.resize(n);
        int score = 0;
        for (int q : a) {
            score += get_score(q, a);
        }
        ld t = 1000, gamma = 0.999;
        ld prev_t = t;
        while (t > 0.001) {
            if (prev_t > 1.1*t) {
                cerr << t << endl;
                prev_t = t;
            }
            int ind = randint() % a.size();
            int x = randint() % (1 << k);
            int delta = delta_score(ind, x, a);
            if (delta >= 0 || P(score, score+delta, t)) {
                a[ind] = x, score += delta;
            }
            t *= gamma;
        }
        return pair{score, a};
    }
```

CHT

```
#define ld long double
const ld E = 1e-8;

struct Line {
    int k, b;

    int value(int x) const {
        return k*x+b;
    }

    ld value(ld x) const {
        return k*x+b;
    }
};

ld inter(Line a, Line b) {
    return (ld)(b.b-a.b)/(a.k-b.k);
}

struct CHT { // max, different angles
    deque<pair<Line, ld>> a;

    void add_increasing(Line line) {
        while (a.size() > 1 && a.back().first.value(a.back().second)-E <
                                line.value(a.back().second)) {
            a.pop_back();
        }
        if (a.empty()) {
            a.emplace_back(line, -INF);
        } else {
            a.emplace_back(line, inter(a.back().first, line));
        }
    }

    void add_decreasing(Line line) {
        while (a.size() > 1 && a[1].first.value(a[1].second)-E < line.value(a[1].second)) {
            a.pop_front();
        }
        if (!a.empty()) {
            a[0].second = inter(a[0].first, line);
        }
        a.emplace_front(line, -INF);
    }

    int ans(int x) const {
        auto lambda = [](pair<Line, ld> x, ld y) {return x.second < y;};
        auto w = --lower_bound(a.begin(), a.end(), x, lambda);
        return w->first.value(x);
    }
};
```

Li_Chao

```
struct Line {
    int k, b;

    int val(int x) {
        return k*x+b;
    }
};

struct Li_Chao {
    vector<Line> a;
    int len = 1;

    Li_Chao(int n) {
        while (len < n) {
            len *= 2;
        }
        a.assign(2*len, {0, INF});
    }

    void descent(int l, int r, int q, Line line) {
        while (q < 2*len) {
            int m = (l+r) >> 1;
            if (a[q].val(m) > line.val(m)) {
                swap(a[q], line);
            }
            if (a[q].val(l) > line.val(l)) {
                r = m;
            } else {
                l = m;
            }
            q *= 2, q += (l == m);
        }
    }

    void add(int l, int r, int L, int R, int q, Line line) {
        if (l >= R || L >= r) {
            return;
        }
        if (L <= l && r <= R) {
            descent(l, r, q, line);
            return;
        }
        int m = (l+r) >> 1;
        add(l, m, L, R, 2*q, line);
        add(m, r, L, R, 2*q+1, line);
    }

    void add(int l, int r, Line line) {
        add(0, len, l, r, 1, line);
    }

    int ans(int x) {
        int q = x+len, res = INF;
        while (q > 0) {
            res = min(res, a[q].val(x));
            q >>= 1;
        }
        return res;
    }
};
```

something_useful

```
__builtin_ffsll(x); // index of the last 1 (0 if x == 0)
__builtin_clzll(x); // number of leading zeros (x != 0)
__builtin_popcountll(x); // number of 1

bitset<17> b;
b._Find_first(); // index of first 1 in bitset
b._Find_next(3); // index of the next 1 in bitset


```

```
#pragma GCC optimize("O3,unroll-loops") // removes short cycles and repeats code of them instead
// [for (int q = 0; q < 3; q++) {ans += q;}] converts to [ans += 1, ans += 2, ans += 3]
#pragma GCC target("avx2,popcnt,lzcnt")
// avx2/avx/sse/sse2/sse3/sse4 -> do multiple operations in parallel in cycles
// popcnt/lzcnt/abm/bmi/bmi2 -> do some bitwise operations in 1 processor action
```

```
set(CMAKE_CXX_FLAGS -fsplit-stack) // lets you create very huge arrays and vectors
set(CMAKE_CXX_FLAGS -fsanitize=address,undefined) // helps to find RE and UB
add_compile_definitions(-D_GLIBCXX_DEBUG -DLOCAL)
```

STL_useful

```
#include <tr2/dynamic_bitset>
typedef tr2::dynamic_bitset<> Bitset;
Bitset a(3);
```

```
#include <bits/extc++.h>
#define int long long
#define ld long double
#define p pair<int, int>
#define endl '\n'
const int INF = (int)1e9+1;

using namespace __gnu_pbds;
using namespace std;

typedef tree<int, null_type, less<>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;

struct my_hash {
    const int seed = chrono::steady_clock::now().time_since_epoch().count();

    static int hash(int x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    int operator()(int x) const {
        return hash(x+seed);
    }
};

gp_hash_table<int, null_type, my_hash> s;
```

Aho_Corasick

```
const int E = 26;
const char FIRST = 'a';
vector<vector<int>> d = {vector<int>(E, -1)};
vector<int> term = {0};

int go_create(int vertex, char w) {
        w -= FIRST;
        if (d[vertex][w] == -1) {
            d[vertex][w] = (int)d.size();
            d.emplace_back(E, -1);
            term.push_back(0);
        }
        return d[vertex][w];
    }

    int add_string(const string& s) {
        int vertex = 0;
        for (char q : s) {
            vertex = go_create(vertex, q);
        }
        term[vertex]++;
        return vertex;
    }

    vector<vector<int>> go;
    vector<int> suf;

    int step(int vertex, int q) {
        if (vertex == -1) {
            return 0;
        }
        int x = d[vertex][q], par = suf[vertex];
        int y = (par == -1 ? 0 : go[par][q]);
        return x == -1 ? y : x;
    }

    void build_suflinks() {
        int m = (int)d.size();
        go = d, suf.assign(m, -1);
        queue<int> a;
        a.push(0);
        while (!a.empty()) {
            int x = a.front();
            a.pop();
            int num = (suf[x] == -1 ? 0 : term[suf[x]]);
            term[x] += num;
            for (int q = 0; q < E; q++) {
                go[x][q] = step(x, q);
                if (d[x][q] != -1) {
                    suf[d[x][q]] = step(suf[x], q);
                    a.push(d[x][q]);
                }
            }
        }
    }
```

Manacher

```
    vector<int> Manacher(vector<int> &a1) {
        int n = a1.size();
        vector<int> a = {a1[0]};
        for (int q = 1; q < n; q++) {
            a.push_back(INF);
            a.push_back(a1[q]);
        }
        n = a.size();
        vector<int> man = {1};
        int ind = 0;
        for (int q = 1; q < n; q++) {
            int k = q;
        if (ind+man[ind] > q) {
            k = min(man[ind]+ind, q+man[2*ind-q]);
        }
        while (k < n && 2*q-k > -1 && a[k] == a[2*q-k]) {
            k++;
        }
        man.push_back(k-q);
        if (k > man[ind]+ind) {
            ind = q;
        }
    }
    for (int q = 0; q < n; q++) {
        if (a[q] == INF) {
            man[q] -= man[q] % 2;
        } else {
            man[q] -= 1-man[q] % 2;
        }
    }
    return man;
}
```

Prefix_function

```
vector<int> pref_func(const string& s) {
    int n = (int)s.size();
    vector<int> pref(n, 0);
    for (int q = 1; q < n; q++) {
        pref[q] = pref[q-1];
        while (pref[q] > 0 && s[pref[q]] != s[q]) {
            pref[q] = pref[pref[q]-1];
        }
        pref[q] += (s[pref[q]] == s[q]);
    }
    return pref;
}
```

Suffix_array

```
vector<int> suf_mas(string &s) {
    s += '#';
    int n = (int)s.size();
    vector<int> suf(n);
    iota(suf.begin(), suf.end(), 0);
    sort(suf.begin(), suf.end(), [&s](int x, int y) {return s[x] < s[y];});
    vector<int> cls(n, 0);
    for (int q = 1; q < n; q++) {
        bool more = (s[suf[q-1]] < s[suf[q]]);
        cls[suf[q]] = cls[suf[q-1]]+more;
    }
    int deg = 1;
    while (cls[suf.back()] < n-1) {
        int num_cls = cls[suf.back()];
        vector<int> nums(num_cls+1, 0);
        for (int q : suf) {
            int ind1 = q-deg+(q < deg)*n;
            nums[cls[ind1]]++;
        }
        vector<int> ind(num_cls+1, 0);
        for (int q = 1; q <= num_cls; q++) {
            ind[q] = ind[q-1]+nums[q-1];
        }
        vector<int> will_suf(n);
        for (int q : suf) {
            int ind1 = q-deg+(q < deg)*n;
            will_suf[ind[cls[ind1]]++] = ind1;
        }
        vector<int> will_cls(n, 0);
        for (int q = 1; q < n; q++) {
            bool start_dif = (cls[will_suf[q-1]] != cls[will_suf[q]]);
            int ind_prev = will_suf[q-1]+deg;
            ind_prev -= (ind_prev >= n)*n;
            int ind_my = will_suf[q]+deg;
            ind_my -= (ind_my >= n)*n;
            bool end_dif = (cls[ind_prev] != cls[ind_my]);
            bool change = (start_dif || end_dif);
            will_cls[will_suf[q]] = will_cls[will_suf[q-1]]+change;
        }
        suf = will_suf, cls = will_cls, deg *= 2;
    }
    return suf;
}

vector<int> LCP(const string& s, const vector<int>& sufmas) {
    int n = (int)s.size()-1; // '#' in the end
    vector<int> pos(n+1), lcp(n+1, 0);
    for (int q = 0; q <= n; q++) {
        pos[sufmas[q]] = q;
    }
    for (int q = 0; q <= n; q++) {
        int q1 = pos[q];
        if (q1 == n) {
            continue;
        }
        int next_q = sufmas[q1+1];
        lcp[q1] = (q == 0 ? 0 : max(0LL, lcp[pos[q-1]] - 1));
        while (s[q+lcp[q1]] == s[next_q+lcp[q1]]) {
            lcp[q1]++;
        }
    }
    lcp.pop_back();
    return lcp;
}
```

Suffix_automaton

```
struct Node {
    int suf = -1, len = 0;
    int term = 0, num = -1;
};

struct Automaton {
    const int E = 10, FIRST = '0';
    vector<vector<int>> d;
    vector<Node> a;
    int end;

    void build_num(int vertex) {
        a[vertex].num = (a[vertex].term > 0);
        for (int q : d[vertex]) {
            if (q != -1 && a[q].num == -1) {
                build_num(q);
            }
            if (q != -1) {
                a[vertex].num += a[q].num;
            }
        }
    }
```

```

        void build_other() {
            while (end != 0) {
                a[end].term = a[end].len-a[a[end].suf].len;
                end = a[end].suf;
            }
            a[0].term = 1;
            build_num(0);
        }

        Automaton() {
            end = add_node();
        }

        explicit Automaton(const string& s): Automaton() {
            for (char q : s) {
                add(q);
            }
            build_other();
        }

        int add_node() {
            d.emplace_back(E, -1);
            a.emplace_back();
            return (int)a.size()-1;
        }

        int clone_node(int q) {
            d.push_back(d[q]);
            a.push_back(a[q]);
            return (int)a.size()-1;
        }

        void add(char w) {
            w -= FIRST;
            int vertex = end;
            end = add_node();
            a[end].len = a[vertex].len+1;
            while (vertex != -1 && d[vertex][w] == -1) {
                d[vertex][w] = end;
                vertex = a[vertex].suf;
            }
            int Q = (vertex == -1 ? 0 : d[vertex][w]);
            if (vertex == -1 || a[Q].len == a[vertex].len+1) {
                a[end].suf = Q;
                return;
            }
            int new_Q = clone_node(Q);
            a[new_Q].len = a[vertex].len+1;
            while (vertex != -1 && d[vertex][w] == Q) {
                d[vertex][w] = new_Q;
                vertex = a[vertex].suf;
            }
            a[Q].suf = a[end].suf = new_Q;
        }

        int go(const string& t) const {
            int vertex = 0;
            for (char q : t) {
                q -= FIRST;
                vertex = d[vertex][q];
                if (vertex == -1) {
                    return -1;
                }
        }
        return vertex;
    }
};
```

Disjoint_Sparse_Table

```
int C;

struct Disjoint_Table {
    vector<vector<int>> s;
    vector<int> level;
    int n, len = 1, log = 0;

    Disjoint_Table(vector<int> a): n(a.size()) {
        while (len <= n) {
            len *= 2, log++;
        }
        a.resize(len, 0);
        s.assign(log, vector<int>(len, 1));
        level = {log};
        for (int q = 1; q < len; q++) {
            level.push_back(level[q >> 1]-1);
        }
        build(0, len, a, 0);
    }

    void build(int l, int r, vector<int>& a, int h) {
        if (r-l == 1) {
            return;
        }
        int m = (l+r) >> 1;
        for (int q = m-1; q >= l; q--) {
            s[h][q] = s[h][q+1]*a[q] % C;
        }
        for (int q = m+1; q < r; q++) {
            s[h][q] = s[h][q-1]*a[q-1] % C;
        }
        build(l, m, a, h+1);
        build(m, r, a, h+1);
    }

    int ans(int l, int r) {
        if (l >= r) {
            return 1;
        }
        int h = level[l ^ r];
        return s[h][l]*s[h][r] % C;
    }
};
```

Fenwick

```
struct Fen {
    vector<int> fen;
    int n;

    Fen(int n1) {
        n = n1+1;
        fen.assign(n, 0);
    }

    void plus(int q, int x) {
        for (++q; q < n; q += (q & -q)) {
            fen[q] += x;
        }
    }

    int sum(int q) {
        int res = 0;
        for (; q > 0; q -= (q & -q)) {
            res += fen[q];
        }
        return res;
    }

    int sum(int l, int r) {
        return sum(r)-sum(l);
    }
};
```

DO_bottom_up

```
template <typename T>
struct DO {
    vector<T> a;
    int n, len = 1;

    T func(T x, T y) {return max(x, y);}
    T I = -INF;

    DO(const vector<T>& init) {
        n = (int)init.size();
        while (len < n) {
            len *= 2;
        }
        a.assign(2*len, I);
        for (int q = len; q < len+n; q++) {
            a[q] = init[q-len];
        }
        for (int q = len-1; q > 0; q--) {
            a[q] = func(a[2*q], a[2*q+1]);
        }
    }

    void change(int q, T x) {
        a[q+len] = x;
        q = (q+len) >> 1;
        while (q > 0) {
            a[q] = func(a[2*q], a[2*q+1]);
            q >>= 1;
        }
    }

    T ans(int l, int r) {
        l += len-1, r += len;
        T res_l = I, res_r = I;
        while (r-l > 1) {
            if ((l & 1) ^ 1) {
                res_l = func(res_l, a[l ^ 1]);
            }
            if (r & 1) {
                res_r = func(a[r ^ 1], res_r);
            }
            l >>= 1, r >>= 1;
        }
        return func(res_l, res_r);
        }

        int right_more(int q, T x) {
            q += len;
            while (q > 0 && ((q & 1) || a[q ^ 1] <= x)) {
                q >>= 1;
            }
            q ^= 1;
            if (a[q] <= x) {
                return n;
            }
            while (q < len) {
                q <<= 1;
                q ^= (a[q] <= x);
            }
            return q-len;
        }
    };
```

Persistent_DO

```
    struct Node {
        int sum = 0;
        int l = 0, r = 0;
    };

    vector<Node> nodes = {Node()};

    int create_node(int ind = 0) {
        nodes.push_back(nodes[ind]);
        return (int)nodes.size()-1;
    }

    void update_DO(int ind) {
        Node& node = nodes[ind];
        node.sum = nodes[node.l].sum+nodes[node.r].sum;
    }

    int build_DO(int n) {
        int ind = create_node();
        if (n != 1) {
            nodes[ind].l = build_DO(n/2);
            nodes[ind].r = build_DO(n-n/2);
        }
        return ind;
    }

    int change_DO(int ind, int l, int r, int q, int x) {
        int new_ind = create_node(ind);
        if (r-l == 1) {
            nodes[new_ind].sum = x;
            return new_ind;
        }
        int m = (l+r) >> 1;
        if (q < m) {
            nodes[new_ind].l = change_DO(nodes[ind].l, l, m, q, x);
        } else {
            nodes[new_ind].r = change_DO(nodes[ind].r, m, r, q, x);
        }
        update_DO(new_ind);
        return new_ind;
    }

    int ans_DO(int ind, int l, int r, int L, int R) {
```

```
    if (l >= R || L >= r) {
        return 0;
    }
    if (L <= l && r <= R) {
        return nodes[ind].sum;
    }
    int m = (l+r) >> 1;
    int ans_l = ans_DO(nodes[ind].l, l, m, L, R);
    int ans_r = ans_DO(nodes[ind].r, m, r, L, R);
    return ans_l+ans_r;
}

struct DO {
    int n, root;

    DO(int n, int ind): n(n), root(ind) {}

    DO change(int q, int x) {
        int new_root = change_DO(root, 0, n, q, x);
        return {n, new_root};
    }

    int ans(int l, int r) {
        return ans_DO(root, 0, n, l, r);
    }
};
```

DD

```
mt19937 randint(17957179);

struct Node {
    static Node* null;

    int x, y, size, sum;
    Node *l, *r;

    Node(int x): x(x), y(randint()), size(1), sum(x) {
        if (null == nullptr) {
            size = 0, null = this;
        }
        l = r = null;
    }
};

Node* Node::null = new Node(0);
Node* null = Node::null;

void update(Node* tree) {
    tree->size = 1, tree->sum = tree->x;
    tree->size += tree->l->size, tree->sum += tree->l->sum;
    tree->size += tree->r->size, tree->sum += tree->r->sum;
}

Node* merge(Node* tree1, Node* tree2) {
    if (tree1 == null || tree2 == null) {
        return tree1 == null ? tree2 : tree1;
    }
    if (tree1->y < tree2->y) {
        tree1->r = merge(tree1->r, tree2);
        update(tree1);
        return tree1;
    }
    tree2->l = merge(tree1, tree2->l);
```

```
        update(tree2);
        return tree2;
    }

    pair<Node*, Node*> split(Node* tree, int x) {
        if (tree == null) {
            return {null, null};
        }
        if (tree->x <= x) {
            auto [left, right] = split(tree->r, x);
            tree->r = left;
            update(tree);
            return {tree, right};
        }
        auto [left, right] = split(tree->l, x);
        tree->l = right;
        update(tree);
        return {left, tree};
    }

    Node* add(Node* tree, int x) {
        Node* node = new Node(x);
        auto [left, right] = split(tree, x);
        return merge(merge(left, node), right);
    }

    Node* del(Node* tree, int x) {
        auto [less_equal, more] = split(tree, x);
        auto [less, equal] = split(less_equal, x-1);
        Node *eq_l = equal->l, *eq_r = equal->r;
        return merge(merge(less, eq_l), merge(eq_r, more));
    }
```

Implicit_DD

```
    mt19937 randint(179);

    struct Node {
        int x, y;
        Node *l, *r, *parent;
        int size, sum;

        Node(int x1): x(x1), y(randint()), l(nullptr), r(nullptr), parent(nullptr), size(1), sum(x) {}
    };

    void update(Node* tree) {
        if (tree == nullptr) {
            return;
        }
        tree->size = 1, tree->sum = tree->x;
        if (tree->l != nullptr) {
            tree->size += tree->l->size, tree->sum += tree->l->sum;
        }
        if (tree->r != nullptr) {
            tree->size += tree->r->size, tree->sum += tree->r->sum;
        }
    }

    void change_parent(Node* tree, Node* parent) {
        if (tree == nullptr) {
            return;
        }
        tree->parent = parent;
    }
```

```
|
|   void change_left(Node* tree, Node* left) {
|       if (tree == nullptr) {
|           return;
|       }
|       tree->l = left;
|       change_parent(left, tree);
|       update(tree);
|   }
|
|   void change_right(Node* tree, Node* right) {
|       if (tree == nullptr) {
|           return;
|       }
|       tree->r = right;
|       change_parent(right, tree);
|       update(tree);
|   }
|
|   Node* merge(Node* tree1, Node* tree2) {
|       if (tree1 == nullptr) {
|           return tree2;
|       }
|       if (tree2 == nullptr) {
|           return tree1;
|       }
|       if (tree1->y < tree2->y) {
|           change_parent(tree1->r, nullptr);
|           change_right(tree1, merge(tree1->r, tree2));
|           return tree1;
|       }
|       change_parent(tree2->l, nullptr);
|       change_left(tree2, merge(tree1, tree2->l));
|       return tree2;
|   }
|
|   pair<Node*, Node*> split(Node* tree, int k) {
|       if (tree == nullptr) {
|           return {nullptr, nullptr};
|       }
|       int t = (tree->l != nullptr ? tree->l->size : 0);
|       if (k <= t) {
|           change_parent(tree->l, nullptr);
|           pair<Node*, Node*> trees = split(tree->l, k);
|           change_left(tree, trees.second);
|           return {trees.first, tree};
|       }
|       change_parent(tree->r, nullptr);
|       pair<Node*, Node*> trees = split(tree->r, k-t-1);
|       change_right(tree, trees.first);
|       return {tree, trees.second};
|   }
|
|   Node* add(Node* tree, int k, Node* vertex) {
|       pair<Node*, Node*> trees = split(tree, k);
|       return merge(merge(trees.first, vertex), trees.second);
|   }
|
|   Node* del(Node* tree, int k) {
|       pair<Node*, Node*> trees1 = split(tree, k);
|       pair<Node*, Node*> trees2 = split(trees1.second, 1);
|       return merge(trees1.first, trees2.second);
|   }
|
Node* root(Node* tree) {
    if (tree == nullptr) {
        return nullptr;
    }
    while (tree->parent != nullptr) {
        tree = tree->parent;
    }
    return tree;
}

int find_pos(Node* tree) {
    if (tree == nullptr) {
        return 0;
    }
    int ans = (tree->l != nullptr ? tree->l->size : 0);
    while (tree->parent != nullptr) {
        if (tree->parent->l != tree) {
            ans += (tree->parent->l != nullptr ? tree->parent->l->size : 0)+1;
        }
        tree = tree->parent;
    }
    return ans;
}

Node* find_element(Node* tree, int k) {
    if (tree == nullptr) {
        return nullptr;
    }
    int t = (tree->l != nullptr ? tree->l->size : 0);
    if (k == t) {
        return tree;
    }
    if (k < t) {
        return find_element(tree->l, k);
    }
    return find_element(tree->r, k-t-1);
}
```

Persistent_DD

```
mt19937 randint(179);

struct Node {
    int x, size;
    Node *l, *r;

    Node(int x1): x(x1), size(1), l(nullptr), r(nullptr) {}
};

void update(Node* tree) {
    if (tree == nullptr) {
        return;
    }
    tree->size = 1;
    if (tree->l != nullptr) {
        tree->size += tree->l->size;
    }
    if (tree->r != nullptr) {
        tree->size += tree->r->size;
    }
}

Node* copy(Node* tree) {
    if (tree == nullptr) {
            return nullptr;
        }
        Node* now = new Node(tree->x);
        now->l = tree->l, now->r = tree->r;
        update(now);
        return now;
    }

    Node* merge(Node* tree1, Node* tree2) {
        if (tree1 == nullptr) {
            return tree2;
        }
        if (tree2 == nullptr) {
            return tree1;
        }
        if (randint() % (tree1->size+tree2->size) < tree1->size) {
            Node* now = copy(tree1);
            now->r = merge(tree1->r, tree2);
            update(now);
            return now;
        }
        Node* now = copy(tree2);
        now->l = merge(tree1, tree2->l);
        update(now);
        return now;
    }

    pair<Node*, Node*> split(Node* tree, int k) {
        if (tree == nullptr) {
            return {nullptr, nullptr};
        }
        int left = (tree->l == nullptr ? 0 : tree->l->size);
        Node* now = copy(tree);
        if (k <= left) {
            pair<Node*, Node*> trees = split(tree->l, k);
            now->l = nullptr;
            update(now);
            trees.second = merge(trees.second, now);
            return trees;
        }
        pair<Node*, Node*> trees = split(tree->r, k-left-1);
        now->r = nullptr;
        update(now);
        trees.first = merge(now, trees.first);
        return trees;
    }

    pair<Node*, bool> change(Node* tree, int pos) {
        auto trees1 = split(tree, pos+1);
        auto trees2 = split(trees1.first, pos);
        Node* will = copy(trees2.second);
        bool flag = (will->x == 1);
        will->x = 1-will->x;
        return {merge(merge(trees2.first, will), trees1.second), flag};
    }
```

ICPC_algorithms

```
|    ./graph/Articulation_points.cpp
|    ./graph/flow/Dinitz.cpp
|    ./graph/flow/Ford_Fulkerson.cpp
|    ./graph/flow/Min_cost.cpp
|    ./graph/Kun.cpp
|    ./graph/tree/HLD.cpp
|    ./graph/tree/LCA/LCA_linear_memory.cpp
|    ./math/Berlekamp.cpp
|    ./math/FFT/FFT_complex.cpp
|    ./math/FFT/FFT_divide.cpp
|    ./math/FFT/FFT_modulo.cpp
|    ./math/FFT/FPS.cpp
|    ./math/FFT/online_FFT.cpp
|    ./math/Floor_sum.cpp
|    ./math/functions/ax_by_c.cpp
|    ./math/functions/C_k_n_any_modulo.cpp
|    ./math/functions/C_k_n_modulo_p.cpp
|    ./math/functions/N_th_root.cpp
|    ./math/functions/Square_root.cpp
|    ./math/geometry/basic.cpp
|    ./math/geometry/polygons.cpp
|    ./math/Grice_Misra_sieve.cpp
|    ./math/matrix/Binary_Gauss.cpp
|    ./math/matrix/SLAE_solve.cpp
|    ./math/Or_And_convolution.cpp
|    ./math/Pollard.cpp
|    ./math/Smiths_theory.cpp
|    ./other/Annealing_simulation.cpp
|    ./other/dp/CHT.cpp
|    ./other/dp/Li_Chao.cpp
|    ./other/something_useful.cpp
|    ./other/STL_useful.cpp
|    ./string/Aho_Corasick.cpp
|    ./string/Manacher.cpp
|    ./string/Prefix_function.cpp
|    ./string/Suffix_array.cpp
|    ./string/Suffix_automaton.cpp
|    ./struct/Disjoint_Sparse_Table.cpp
|    ./struct/Fenwick.cpp
|    ./struct/Segment_Tree/DO_bottom_up.cpp
|    ./struct/Segment_Tree/Persistent_DO.cpp
|    ./struct/Treap/DD.cpp
|    ./struct/Treap/Implicit_DD.cpp
```